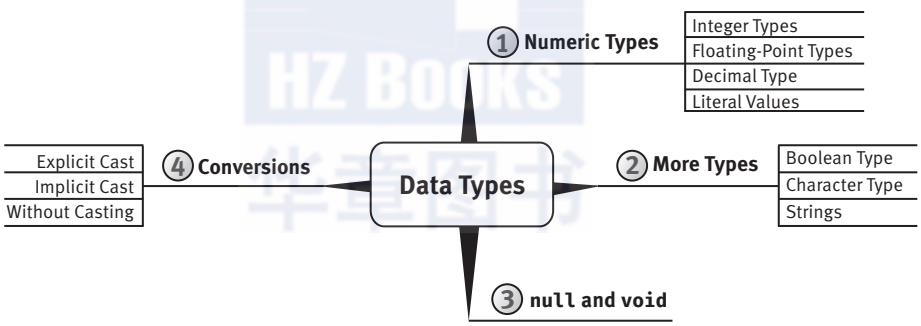
2 Data Types

From Chapter 1's `HelloWorld` program, you got a feel for the C# language, its structure, basic syntax characteristics, and how to write the simplest of programs. This chapter continues to discuss the C# basics by investigating the fundamental C# types.



Until now, you have worked with only a few built-in data types, with little explanation. In C# thousands of types exist, and you can combine types to create new types. A few types in C#, however, are relatively simple and are considered the building blocks of all other types. These types are the **predefined types**. The C# language's predefined types include eight integer types, two binary floating-point types for scientific calculations and one decimal float for financial calculations, one Boolean type, and a character type. This chapter investigates these types and looks more closely at the `string` type.

Fundamental Numeric Types

The basic numeric types in C# have keywords associated with them. These types include integer types, floating-point types, and a special floating-point type called `decimal` to store large numbers with no representation error.

Integer Types

There are eight C# integer types. This variety allows you to select a data type large enough to hold its intended range of values without wasting resources. Table 2.1 lists each integer type.

TABLE 2.1: Integer Types

Type	Size	Range (Inclusive)	BCL Name	Signed	Literal Suffix
`sbyte`	8 bits	–128 to 127	`System.SByte`	Yes	
`byte`	8 bits	0 to 255	`System.Byte`	No	
`short`	16 bits	–32,768 to 32,767	`System.Int16`	Yes	
`ushort`	16 bits	0 to 65,535	`System.UInt16`	No	
`int`	32 bits	–2,147,483,648 to 2,147,483,647	`System.Int32`	Yes	
`uint`	32 bits	0 to 4,294,967,295	`System.UInt32`	No	`U` or `u`
`long`	64 bits	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	`System.Int64`	Yes	`L` or `l`
`ulong`	64 bits	0 to 18,446,744,073,709,551,615	`System.UInt64`	No	`UL` or `ul`

Included in Table 2.1 (and in Tables 2.2 and 2.3) is a column for the full name of each type; we discuss the literal suffix later in the chapter. All the fundamental types in C# have both a short name and a full name. The full name corresponds to the type as it is named in the Base Class Library (BCL). This name, which is the same across all languages, uniquely identifies the type within an assembly. Because of the fundamental nature of these types, C# also supplies keywords as short names or abbreviations to the full names of fundamental types. From the compiler's perspective,

both names refer to the same type, producing identical code. In fact, an examination of the resultant Common Intermediate Language (CIL) code would provide no indication of which name was used.

Although C# supports using both the full BCL name and the keyword, as developers we are left with the choice of which to use when. Rather than switching back and forth, it is better to use one or the other consistently. For this reason, C# developers generally use the C# keyword form—choosing, for example, `int` rather than `System.Int32` and `string` rather than `System.String` (or a possible shortcut of `String`).

Guidelines

DO use the C# keyword rather than the BCL name when specifying a data type (e.g., **`string`** rather than **`String`**).

DO favor consistency rather than variety within your code.

The choice for consistency frequently may be at odds with other guidelines. For example, given the guideline to use the C# keyword in place of the BCL name, there may be occasions when you find yourself maintaining a file (or library of files) with the opposite style. In these cases, it would be better to stay consistent with the previous style than to inject a new style and inconsistencies in the conventions. Even so, if the "style" was a bad coding practice that was likely to introduce bugs and obstruct successful maintenance, by all means correct the issue throughout.

Language Contrast: C++—`short` Data Type

In C/C++, the `short` data type is an abbreviation for `short int`. In C#, `short` on its own is the actual data type.

Floating-Point Types (`float`, `double`)

Floating-point numbers have varying degrees of precision, and binary floating-point types can represent numbers exactly only if they are a fraction with a power of 2 as the denominator. If you were to set the value of a floating-point variable to be 0.1, it could very easily be represented as 0.0999999999999999 or 0.10000000000000001 or some other number

very close to 0.1. Similarly, setting a variable to a large number such as Avogadro's number, 6.02×10^{23}, could lead to a representation error of approximately 10^{8}, which after all is a tiny fraction of that number. The accuracy of a floating-point number is in proportion to the magnitude of the number it represents. A floating-point number is precise to a certain number of significant digits, not by a fixed value such as ±0.01.

C# supports the two binary floating-point number types listed in Table 2.2.

TABLE 2.2: Floating-Point Types

Type	Size	Range (Inclusive)	BCL Name	Significant Digits	Literal Suffix
`float`	32 bits	$\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$	`System.Single`	7	`F` or `f`
`double`	64 bits	$\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$	`System.Double`	15–16	

Binary numbers appear as base 10 (denary) numbers for human readability. The number of bits (binary digits) converts to 15 decimal digits, with a remainder that contributes to a sixteenth decimal digit as expressed in Table 2.2. Specifically, numbers between 1.7×10^{307} and less than 1×10^{308} have only 15 significant digits. However, numbers ranging from 1×10^{308} to 1.7×10^{308} will have 16 significant digits. A similar range of significant digits occurs with the `decimal` type as well.

Decimal Type

C# also provides a decimal floating-point type with 128-bit precision (see Table 2.3). This type is suitable for financial calculations.

TABLE 2.3: Decimal Type

Type	Size	Range (Inclusive)	BCL Name	Significant Digits	Literal Suffix
`decimal`	128 bits	1.0×10^{-28} to approximately 7.9×10^{28}	`System.Decimal`	28–29	`M` or `m`

Unlike binary floating-point numbers, the `decimal` type maintains exact accuracy for all denary numbers within its range. With the `decimal` type, therefore, a value of 0.1 is exactly 0.1. However, while the `decimal` type has greater precision than the floating-point types, it has a smaller range. Thus, conversions from floating-point types to the `decimal` type may result in overflow errors. Also, calculations with `decimal` are slightly (generally imperceptibly) slower.

ADVANCED TOPIC

Floating-Point Types Dissected

Denary numbers within the range and precision limits of the `decimal` type are represented exactly. In contrast, the binary floating-point representation of many denary numbers introduces a rounding error. Just as ⅓ cannot be represented exactly in any finite number of decimal digits, so $^{11}/_{10}$ cannot be represented exactly in any finite number of binary digits. In both cases, we end up with a rounding error of some kind.

A `decimal` is represented by $\pm N * 10^k$ where the following is true:

- `N`, the mantissa, is a positive 96-bit integer.
- `k`, the exponent, is given by `-28 <= k <= 0`.

In contrast, a binary float is any number $\pm N * 2^k$ where the following is true:

- `N` is a positive 24-bit (for `float`) or 53-bit (for `double`) integer.
- `k` is an integer ranging from `-149` to `+104` for `float` and from `-1074` to `+970` for `double`.

Literal Values

A **literal value** is a representation of a constant value within source code. For example, if you want to have `System.Console.WriteLine()` print out the integer value `42` and the `double` value `1.618034`, you could use the code shown in Listing 2.1.

Listing 2.1: Specifying Literal Values

```
System.Console.WriteLine(42);
System.Console.WriteLine(1.618034);
```

Output 2.1 shows the results of Listing 2.1.

Output 2.1

```
42
1.618034
```

BEGINNER TOPIC

Use Caution When Hardcoding Values

The practice of placing a value directly into source code is called **hardcoding**, because changing the values requires recompiling the code. Developers must carefully consider the choice between hardcoding values within their code and retrieving them from an external source, such as a configuration file, so that the values are modifiable without recompiling.

By default, when you specify a literal number with a decimal point, the compiler interprets it as a `double` type. Conversely, a literal value with no decimal point generally defaults to an `int`, assuming the value is not too large to be stored in an integer. If the value is too large, the compiler will interpret it as a `long`. Furthermore, the C# compiler allows assignment to a numeric type other than an `int`, assuming the literal value is appropriate for the target data type. `short s = 42` and `byte b = 77` are allowed, for example. However, this is appropriate only for constant values; `b = s` is not allowed without additional syntax, as discussed in the section "Conversions between Data Types" later in this chapter.

As previously discussed in this section, there are many different numeric types in C#. In Listing 2.2, a literal value is placed within C# code. Since numbers with a decimal point will default to the `double` data type, the output, shown in Output 2.2, is `1.61803398874989` (the last digit, 5, is missing), corresponding to the expected accuracy of a `double`.

Listing 2.2: Specifying a Literal double

```
System.Console.WriteLine(1.618033988749895);
```

Output 2.2

```
1.61803398874989
```

To view the intended number with its full accuracy, you must declare explicitly the literal value as a `decimal` type by appending an `M` (or `m`) (see Listing 2.3 and Output 2.3).

Listing 2.3: Specifying a Literal decimal

```
System.Console.WriteLine(1.618033988749895M);
```

Output 2.3

```
1.618033988749895
```

Now the output of Listing 2.3 is as expected: `1.618033988749895`. Note that `d` is the abbreviation for `double`. To remember that `m` should be used to identify a `decimal`, remember that "`m` is for monetary calculations."

You can also add a suffix to a value to explicitly declare a literal as a `float` or `double` by using the `F` and `D` suffixes, respectively. For integer data types, the suffixes are `U`, `L`, `LU`, and `UL`. The type of an integer literal can be determined as follows:

- Numeric literals with no suffix resolve to the first data type that can store the value, in this order: `int`, `uint`, `long`, and `ulong`.
- Numeric literals with the suffix `U` resolve to the first data type that can store the value, in the order `uint` and then `ulong`.
- Numeric literals with the suffix `L` resolve to the first data type that can store the value, in the order `long` and then `ulong`.
- If the numeric literal has the suffix `UL` or `LU`, it is of type `ulong`.

Note that suffixes for literals are case insensitive. However, uppercase is generally preferred to avoid any ambiguity between the lowercase letter `l` and the digit `1`.

Guidelines

DO use uppercase literal suffixes (e.g., `1.618033988749895M`).

Begin 7.0

On occasion, numbers can get quite large and difficult to read. To overcome the readability problem, C# 7.0 added support for a **digit separator**, an underscore (_), when expressing a numeric literal, as shown in Listing 2.4.

Listing 2.4: Specifying Digit Separator

```
System.Console.WriteLine(9_814_072_356);
```

In this case, we separate the digits into thousands (threes), but this is not required by C#. You can use the digit separator to create whatever grouping you like as long as the underscore occurs between the first and the last digit. In fact, you can even have multiple underscores side by side—with no digit between them.

End 7.0

In addition, you may wish to use exponential notation instead of writing out several zeroes before or after the decimal point (whether using a digit separator or not). To use exponential notation, supply the `e` or `E` infix, follow the infix character with a positive or negative integer number, and complete the literal with the appropriate data type suffix. For example, you could print out Avogadro's number as a `float`, as shown in Listing 2.5 and Output 2.4.

Listing 2.5: Exponential Notation

```
System.Console.WriteLine(6.023E23F);
```

Output 2.4

```
6.023E+23
```


BEGINNER TOPIC

Hexadecimal Notation

Usually you work with numbers that are represented with a base of 10, meaning there are 10 symbols (0–9) for each place value in the number. If

a number is displayed with hexadecimal notation, it is displayed with a base of 16 numbers, meaning 16 symbols are used: 0–9, A–F (lowercase can also be used). Therefore, 0x000A corresponds to the decimal value 10 and 0x002A corresponds to the decimal value 42, being 2 × 16 + 10. The actual number is the same. Switching from hexadecimal to decimal, or vice versa, does not change the number itself, just the representation of the number.

Each hex digit is four bits, so a byte can represent two hex digits.

In all discussions of literal numeric values so far, we have covered only base 10 type values. C# also supports the ability to specify hexadecimal values. To specify a hexadecimal value, prefix the value with `0x` and then use any hexadecimal digit, as shown in Listing 2.6.

Listing 2.6: Hexadecimal Literal Value

```
// Display the value 42 using a hexadecimal literal
System.Console.WriteLine(0x002A);
```

Output 2.5 shows the results of Listing 2.6.

Output 2.5

```
42
```

Note that this code still displays `42`, not `0x002A`.

Starting with C# 7.0, you can also represent numbers as binary values (see Listing 2.7).

Begin 7.0

Listing 2.7: Binary Literal Value

```
// Display the value 42 using a binary literal
System.Console.WriteLine(0b101010);
```

The syntax is like the hexadecimal syntax except with a `0b` as the prefix (an uppercase `B` is also allowed). See the Beginner Topic titled "Bits and Bytes" in Chapter 4 for a full explanation of binary notation and the conversion between binary and decimal.

Note that starting with C# 7.2, you can place the digit separator after the `x` for a hexadecimal literal or the `b` for a binary literal.

End 7.0

ADVANCED TOPIC

Formatting Numbers as Hexadecimal

To display a numeric value in its hexadecimal format, it is necessary to use the `x` or `X` numeric formatting specifier. The casing determines whether the hexadecimal letters appear in lowercase or uppercase. Listing 2.8 shows an example of how to do this.

Listing 2.8: Example of a Hexadecimal Format Specifier

```
// Displays "0x2A"
System.Console.WriteLine($"0x{42:X}");
```

Output 2.6 shows the results.

Output 2.6

```
0x2A
```

Note that the numeric literal (`42`) can be in decimal or hexadecimal form. The result will be the same. Also, to achieve the hexadecimal formatting, we rely on the formatting specifier, separated from the string interpolation expression with a colon.

ADVANCED TOPIC

Round-Trip Formatting

By default, `System.Console.WriteLine(1.618033988749895);` displays `1.61803398874989`, with the last digit missing. To more accurately identify the string representation of the double value, it is possible to convert it using a format string and the round-trip format specifier, `R` (or `r`). For example, `string.Format("{0:R}", 1.618033988749895)` will return the result `1.6180339887498949`.

The round-trip format specifier returns a string that, if converted back into a numeric value, will always result in the original value. Listing 2.9 shows the numbers are not equal without use of the round-trip format.

LISTING 2.9: Formatting Using the R Format Specifier

```
// ...
const double number = 1.618033988749895;
double result;
string text;

text = $"{number}";
result = double.Parse(text);
System.Console.WriteLine($"{result == number}: result == number");

text = string.Format("{0:R}", number);
result = double.Parse(text);
System.Console.WriteLine($"{result == number}: result == number");

// ...
```

Output 2.7 shows the resultant output.

OUTPUT 2.7

```
False: result == number
True: result == number
```

When assigning `text` the first time, there is no round-trip format specifier; as a result, the value returned by `double.Parse(text)` is not the same as the original `number` value. In contrast, when the round-trip format specifier is used, `double.Parse(text)` returns the original value.

For those readers who are unfamiliar with the `==` syntax from C-based languages, `result == number` evaluates to `true` if `result` is equal to `number`, while `result != number` does the opposite. Both assignment and equality operators are discussed in the next chapter.

More Fundamental Types

The fundamental types discussed so far are numeric types. C# includes some additional types as well: `bool`, `char`, and `string`.

Boolean Type (bool)

Another C# primitive is a Boolean or conditional type, `bool`, which represents true or false in conditional statements and expressions. Allowable

values are the keywords `true` and `false`. The BCL name for `bool` is `System.Boolean`. For example, to compare two strings in a case-insensitive manner, you call the `string.Compare()` method and pass a `bool` literal `true` (see Listing 2.10).

Listing 2.10: A Case-Insensitive Comparison of Two Strings

```
string option;
...
int comparison = string.Compare(option, "/Help", true);
```

In this case, you make a case-insensitive comparison of the contents of the variable `option` with the literal text `/Help` and assign the result to `comparison`.

Although theoretically a single bit could hold the value of a Boolean, the size of `bool` is 1 byte.

Character Type (char)

A `char` type represents 16-bit characters whose set of possible values are drawn from the Unicode character set's UTF-16 encoding. A `char` is the same size as a 16-bit unsigned integer (`ushort`), which represents values between 0 and 65,535. However, `char` is a unique type in C# and code should treat it as such.

The BCL name for `char` is `System.Char`.

BEGINNER TOPIC

The Unicode Standard

Unicode is an international standard for representing characters found in most human languages. It provides computer systems with functionality for building **localized** applications—that is, applications that display the appropriate language and culture characteristics for different cultures.

ADVANCED TOPIC

16 Bits Is Too Small for All Unicode Characters

Unfortunately, not all Unicode characters can be represented by just one 16-bit `char`. The original Unicode designers believed that 16 bits would be

enough, but as more languages were supported, it was realized that this assumption was incorrect. As a result, some (rarely used) Unicode characters are composed of "surrogate pairs" of two `char` values.

To construct a literal `char`, place the character within single quotes, as in `'A'`. Allowable characters comprise the full range of keyboard characters, including letters, numbers, and special symbols.

Some characters cannot be placed directly into the source code and instead require special handling. These characters are prefixed with a backslash (\) followed by a special character code. In combination, the backslash and special character code constitute an **escape sequence**. For example, `\n` represents a newline, and `\t` represents a tab. Since a backslash indicates the beginning of an escape sequence, it can no longer identify a simple backslash; instead, you need to use `\\` to represent a single backslash character.

Listing 2.11 writes out one single quote because the character represented by `\'` corresponds to a single quote.

LISTING 2.11: Displaying a Single Quote Using an Escape Sequence

```
class SingleQuote
{
  static void Main()
  {
      System.Console.WriteLine('\'');
  }
}
```

In addition to showing the escape sequences, Table 2.4 includes the Unicode representation of characters.

TABLE 2.4: Escape Characters

Escape Sequence	Character Name	Unicode Encoding
`\'`	Single quote	`\u0027`
`\"`	Double quote	`\u0022`
`\\`	Backslash	`\u005C`
`\0`	Null	`\u0000`

continues

TABLE 2.4: Escape Characters (*continued*)

Escape Sequence	Character Name	Unicode Encoding
`\a`	Alert (system beep)	`\u0007`
`\b`	Backspace	`\u0008`
`\f`	Form feed	`\u000C`
`\n`	Line feed (sometimes referred to as a newline)	`\u000A`
`\r`	Carriage return	`\u000D`
`\t`	Horizontal tab	`\u0009`
`\v`	Vertical tab	`\u000B`
`\uxxxx`	Unicode character in hex	`\u0029`
`\x[n][n][n]n`	Unicode character in hex (first three placeholders are options); variable-length version of `\uxxxx`	`\u3A`
`\Uxxxxxxxx`	Unicode escape sequence for creating surrogate pairs	`\UD840DC01` (𠀁)

You can represent any character using Unicode encoding. To do so, prefix the Unicode value with `\u`. You represent Unicode characters in hexadecimal notation. The letter `A`, for example, is the hexadecimal value `0x41`. Listing 2.12 uses Unicode characters to display a smiley face (`:)`), and Output 2.8 shows the results.

LISTING 2.12: Using Unicode Encoding to Display a Smiley Face

```
System.Console.Write('\u003A');
System.Console.WriteLine('\u0029');
```

OUTPUT 2.8

```
:)
```

Strings

A finite sequence of zero or more characters is called a **string**. The string type in C# is `string`, whose BCL name is `System.String`. The string type includes some special characteristics that may be unexpected to developers familiar with other programming languages. In addition to the string literal format discussed in Chapter 1, strings include a "verbatim string" prefix character of `@`, support for string interpolation with the `$` prefix character, and the potentially surprising fact that strings are immutable.

Literals

You can enter a literal string into code by placing the text in double quotes ("), as you saw in the `HelloWorld` program. Strings are composed of characters, and consequently, character escape sequences can be embedded within a string.

In Listing 2.13, for example, two lines of text are displayed. However, instead of using `System.Console.WriteLine()`, the code listing shows `System.Console.Write()` with the newline character, `\n`. Output 2.9 shows the results.

Listing 2.13: Using the \n Character to Insert a Newline

```
class DuelOfWits
{
  static void Main()
  {
      System.Console.Write(
          "\"Truly, you have a dizzying intellect.\"");
      System.Console.Write("\n\"Wait 'til I get going!\"\n");
  }
}
```

Output 2.9

```
"Truly, you have a dizzying intellect."
"Wait 'til I get going!"
```

The escape sequence for double quotes differentiates the printed double quotes from the double quotes that define the beginning and end of the string.

In C#, you can use the @ symbol in front of a string to signify that a backslash should not be interpreted as the beginning of an escape sequence. The resultant **verbatim string literal** does not reinterpret just the backslash character. Whitespace is also taken verbatim when using the @ string syntax. The triangle in Listing 2.14, for example, appears in the console exactly as typed, including the backslashes, newlines, and indentation. Output 2.10 shows the results.

LISTING 2.14: Displaying a Triangle Using a Verbatim String Literal

```
class Triangle
{
  static void Main()
  {
      System.Console.Write(@"begin
             /\
            /  \
           /    \
          /      \
         /________\
end");
  }
}
```

OUTPUT 2.10

```
begin
             /\
            /  \
           /    \
          /      \
         /________\
end
```

Without the @ character, this code would not even compile. In fact, even if you changed the shape to a square, eliminating the backslashes, the code still would not compile because a newline cannot be placed directly within a string that is not prefaced with the @ symbol.

The only escape sequence the verbatim string does support is "", which signifies double quotes and does not terminate the string.

Language Contrast: C++—String Concatenation at Compile Time

Unlike C++, C# does not automatically concatenate literal strings. You cannot, for example, specify a string literal as follows:

```
"Major Strasser has been shot."
"Round up the usual suspects."
```

Rather, concatenation requires the use of the addition operator. (If the compiler can calculate the result at compile time, however, the resultant CIL code will be a single string.)

If the same literal string appears within an assembly multiple times, the compiler will define the string only once within the assembly and all variables will refer to the same string. That way, if the same string literal containing thousands of characters was placed multiple times into the code, the resultant assembly would reflect the size of only one of them.

String Interpolation

Begin 6.0

As discussed in Chapter 1, strings can support embedded expressions when using the string interpolation format starting in C# 6.0. The string interpolation syntax prefixes a verbatim string literal with a dollar symbol and then embeds the expressions within curly brackets. The following is an example:

```
System.Console.WriteLine($"Your full name is {firstName} {lastName}.");
```

where `firstName` and `lastName` are simple expressions that refer to variables. Note that verbatim string literals can be combined with string interpolation by specifying the `$` prior to the `@` symbol, as in this example:

```
System.Console.WriteLine($@"Your full name is:
    { firstName } { lastName }");
```

Since this is a verbatim string literal, the text is output on two lines. You can, however, make a similar line break in the code without incurring a line break in the output by placing the line feeds inside the curly braces as follows:

```
System.Console.WriteLine($@"Your full name is: {
    firstName } { lastName }");
```

Note that the @ symbol is still required even when only placing the new lines within the curly braces.

ADVANCED TOPIC

Understanding the Internals of String Interpolation

String interpolation is a shorthand for invoking the `string.Format()` method. For example, a statement such as

```
System.Console.WriteLine($"Your full name is {firstName} {lastName}.")
```

will be transformed to the C# equivalent of

```
object[] args = new object[] { firstName, lastName };
Console.WriteLine(string.Format("Your full name is {0} {1}.", args));
```

This leaves in place support for localization in the same way it works with composite string and doesn't introduce any post-compile injection of code via strings.

End 6.0

String Methods

The `string` type, like the `System.Console` type, includes several methods. There are methods, for example, for formatting, concatenating, and comparing strings.

The `Format()` method in Table 2.5 behaves similarly to the `Console.Write()` and `Console.WriteLine()` methods, except that instead of displaying the result in the console window, `string.Format()` returns the result to the caller. Of course, with string interpolation the need for `string.Format()` is significantly reduced (except for localization support). Under the covers, however, string interpolation compiles down to CIL that leverages `string.Format()`.

All of the methods in Table 2.5 are **static**. This means that, to call the method, it is necessary to prefix the method name (e.g., `Concat`) with the type that contains the method (e.g., `string`). As illustrated later in this chapter, however, some of the methods in the string class are **instance** methods. Instead of prefixing the method with the type, instance methods use the variable name (or some other reference to an instance). Table 2.6 shows a few of these methods, along with an example.

TABLE 2.5: string Static Methods

Statement	Example

static string string.
Format(
  string format,
  ...)
	string text, firstName, lastName;
	//...
	text = **string**.Format("Your full name is {0} {1}.",
	  firstName, lastName);
	// Display
	// "Your full name is <firstName> <lastName>."
	System.Console.WriteLine(text);

static string string.
Concat(
  string str0,
  string str1)
	string text, firstName, lastName;
	//...
	text = string.Concat(firstName, lastName);
	// Display "<firstName><lastName>", notice
	// that there is no space between names
	System.Console.WriteLine(text);

static int string.
Compare(
  string str0,
  string str1)
	string option;
	//...
	// String comparison in which case matters
	int result = **string**.Compare(option, "/help");
	
	// Display:
	// 0 if equal
	// negative if option < /help
	// positive if option > /help
	System.Console.WriteLine(result);

	string option;
	//...
	// Case-insensitive string comparison
	int result = **string**.Compare(
	  option, "/Help", **true**);
	
	// Display:
	// 0 if equal
	// < 0 if option < /help
	// > 0 if option > /help
	System.Console.WriteLine(result);

TABLE 2.6: string Methods

Statement	Example

bool StartsWith(
  string value)
bool EndsWith(
  string value)
	string lastName
	//...
	bool isPhd = lastName.EndsWith("Ph.D.");
	bool isDr = lastName.StartsWith("Dr.");

continues

TABLE 2.6: string Methods (*continued*)

Statement	Example

`string ToLower()`
`string ToUpper()`
	`string severity = "warning";`
	`// Display the severity in uppercase`
	`System.Console.WriteLine(severity.ToUpper());`

`string Trim()`
`string Trim(...)`
`string TrimEnd()`
`string TrimStart()`
	`// Remove any whitespace at the start or end`
	`username = username.Trim();`

`string Replace(`
`  string oldValue,`
`  string newValue)`
	`string filename;`
	`//...`
	`// Remove ?'s from the string`
	`filename = filename.Replace("?", "");;`

ADVANCED TOPIC

using Directive and using static Directive

Begin 6.0

The invocation of static methods as we have used them so far always involves a prefix of the namespace followed by the type name. When calling `System.Console.WriteLine`, for example, even though the method invoked is `WriteLine()` and there is no other method with that name within the context, it is still necessary to prefix the method name with the namespace (`System`) followed by the type name (`Console`). On occasion, you may want a shortcut to avoid such explicitness; to do so, you can leverage the C# 6.0 `using static` directive, as shown in Listing 2.15.

LISTING 2.15: using static Directive

```
// The using directives allow you to drop the namespace
using static System.Console;
class HeyYou
{
  static void Main()
  {
      string firstName;
      string lastName;

      WriteLine("Hey you!");

      Write("Enter your first name: ");
      firstName = ReadLine();
```

```
        Write("Enter your last name: ");
        lastName = ReadLine();

        WriteLine(
          $"Your full name is {firstName} {lastName}.");
    }
}
```

The `using static` directive needs to appear at the top of the file.[1] Each time we use the `System.Console` class, it is no longer necessary to also use the `System.Console` prefix. Instead, we can simply write the method name. An important point to note about the `using static` directive is that it works only for static methods and properties, not for instance members.

A similar directive, the `using` directive, allows for eliminating the namespace prefix—for example, `System`. Unlike the `using static` directive, the `using` directive applies universally within the file (or namespace) in which it resides (not just to static members). With the `using` directive, you can (optionally) eliminate all references to the namespace, whether during instantiation, during static method invocation, or even with the `nameof` operator found in C# 6.0.

End 6.0

String Formatting

Whether you use `string.Format()` or the C# 6.0 string interpolation feature to construct complex formatting strings, a rich and complex set of formatting patterns is available to display numbers, dates, times, timespans, and so on. For example, if `price` is a variable of type `decimal`, then `string.Format("{0,20:C2}", price)` or the equivalent interpolation `$"{price,20:C2}"` both convert the decimal value to a string using the default currency formatting rules, rounded to two figures after the decimal place and right-justified in a 20-character-wide string. Space does not permit a detailed discussion of all the possible formatting strings; consult the MSDN documentation for `string.Format()` for a complete listing of formatting strings.

If you want an actual left or right curly brace inside an interpolated string or formatted string, you can double the brace to indicate that it is not introducing a pattern. For example, the interpolated string `$"{{ {price:C2} }}"` might produce the string `"{ $1,234.56 }"`

1. Or at the top of a namespace declaration.

Newline

When writing out a newline, the exact characters for the newline will depend on the operating system on which you are executing. On Microsoft Windows operating systems, the newline is the combination of both the carriage return (`\r`) and line feed (`\n`) characters, while a single line feed is used on UNIX. One way to overcome the discrepancy between operating systems is simply to use `System.Console.WriteLine()` to output a blank line. Another approach, which is almost essential for working with newlines from the same code base on multiple operating systems, is to use `System.Environment.NewLine`. In other words, `System.Console.WriteLine("Hello World")` and `System.Console.Write($"Hello World{System.Environment.NewLine}")` are equivalent. However, on Windows, `System.WriteLine()` and `System.Console.Write(System.Environment.NewLine)` are equivalent to `System.Console.Write("\r\n")`—not `System.Console.Write("\n")`. In summary, rely on `System.WriteLine()` and `System.Environment.NewLine` rather than `\n` to accommodate Windows-specific operating system idiosyncrasies with the same code that runs on Linux and iOS.

Guidelines

DO rely on `System.WriteLine()` and `System.Environment.NewLine` rather than `\n` to accommodate Windows-specific operating system idiosyncrasies with the same code that runs on Linux and iOS.

ADVANCED TOPIC

C# Properties

The `Length` member referred to in the following section is not actually a method, as indicated by the fact that there are no parentheses following its call. `Length` is a property of `string`, and C# syntax allows access to a property as though it were a member variable (known in C# as a **field**). In other words, a property has the behavior of special methods called setters and getters, but the syntax for accessing that behavior is that of a field.

Examining the underlying CIL implementation of a property reveals that it compiles into two methods: `set_<PropertyName>` and

get_<PropertyName>. Neither of these, however, is directly accessible from C# code, except through the C# property constructs. See Chapter 6 for more details on properties.

String Length

To determine the length of a string, you use a string member called Length. This particular member is called a **read-only property**. As such, it cannot be set, nor does calling it require any parameters. Listing 2.16 demonstrates how to use the Length property, and Output 2.11 shows the results.

LISTING 2.16: Using string's Length Member

```
class PalindromeLength
{
  static void Main()
  {
      string palindrome;

      System.Console.Write("Enter a palindrome: ");
      palindrome = System.Console.ReadLine();

      System.Console.WriteLine(
          $"The palindrome \"{palindrome}\" is"
          + $" {palindrome.Length} characters.");
  }
}
```

OUTPUT 2.11

```
Enter a palindrome: Never odd or even
The palindrome "Never odd or even" is 17 characters.
```

The length for a string cannot be set directly; it is calculated from the number of characters in the string. Furthermore, the length of a string cannot change because a string is **immutable**.

Strings Are Immutable

A key characteristic of the string type is that it is immutable. A string variable can be assigned an entirely new value, but there is no facility for modifying the contents of a string. It is not possible, therefore, to convert a string to all uppercase letters. It is trivial to create a new string that is composed of an uppercase version of the old string, but the old string is not modified in the process. Consider Listing 2.17 as an example.

LISTING 2.17: Error; `string` Is Immutable

```
class Uppercase
{
  static void Main()
  {
      string text;

      System.Console.Write("Enter text: ");
      text = System.Console.ReadLine();

      // UNEXPECTED:  Does not convert text to uppercase
      text.ToUpper();

      System.Console.WriteLine(text);
  }
}
```

Output 2.12 shows the results of Listing 2.17.

OUTPUT 2.12

```
Enter text: This is a test of the emergency broadcast system.
This is a test of the emergency broadcast system.
```

At a glance, it would appear that `text.ToUpper()` should convert the characters within `text` to uppercase. However, strings are immutable and, therefore, `text.ToUpper()` will make no such modification. Instead, `text.ToUpper()` returns a new string that needs to be saved into a variable or passed to `System.Console.WriteLine()` directly. The corrected code is shown in Listing 2.18, and its output is shown in Output 2.13.

LISTING 2.18: Working with Strings

```
class Uppercase
{
  static void Main()
  {
      string text, uppercase;

      System.Console.Write("Enter text: ");
      text = System.Console.ReadLine();

      // Return a new string in uppercase
      uppercase = text.ToUpper();

      System.Console.WriteLine(uppercase);
  }
}
```

OUTPUT 2.13

```
Enter text: This is a test of the emergency broadcast system.
THIS IS A TEST OF THE EMERGENCY BROADCAST SYSTEM.
```

If the immutability of a string is ignored, mistakes like those shown in Listing 2.17 can occur with other string methods as well.

To actually change the value of `text`, assign the value from `ToUpper()` back into `text`, as in the following code:

```
text = text.ToUpper();
```

System.Text.StringBuilder

If considerable string modification is needed, such as when constructing a long string in multiple steps, you should use the data type `System.Text.StringBuilder` rather than `string`. The `StringBuilder` type includes methods such as `Append()`, `AppendFormat()`, `Insert()`, `Remove()`, and `Replace()`, some of which are also available with `string`. The key difference, however, is that with `StringBuilder` these methods will modify the data in the `StringBuilder` itself and will not simply return a new string.

null and void

Two additional keywords relating to types are `null` and `void`. The `null` value, identified with the `null` keyword, indicates that the variable does not refer to any valid object. `void` is used to indicate the absence of a type or the absence of any value altogether.

null

`null` can also be used as a type of string "literal." `null` indicates that a variable is set to nothing. Reference types, pointer types, and nullable value types can be assigned the value `null`. The only reference type covered so far in this book is `string`; Chapter 6 covers the topic of creating classes (which are reference types) in detail. For now, suffice it to say that a variable of reference type contains a reference to a location in memory that is different from the value of the variable. Code that sets a variable to `null` explicitly assigns the reference to refer to no valid value. In fact, it is even possible to check whether a reference refers to nothing. Listing 2.19 demonstrates assigning `null` to a `string` variable.

LISTING 2.19: Assigning `null` to a String

```
static void Main()
{
    string faxNumber;
    // ...

    // Clear the value of faxNumber
    faxNumber = null;

    // ...
}
```

Assigning the value `null` to a reference type is not equivalent to not assigning it at all. In other words, a variable that has been assigned `null` has still been set, whereas a variable with no assignment has not been set and therefore will often cause a compile error if used prior to assignment.

Assigning the value `null` to a `string` variable is distinctly different from assigning an empty string, `""`. Use of `null` indicates that the variable has no value, whereas `""` indicates that there is a value—an empty string. This type of distinction can be quite useful. For example, the programming logic could interpret a `homePhone` of `null` to mean that the home phone number is unknown, while a `homePhone` value of `""` could indicate that there is no home phone number.

The `void` "Type"

Sometimes the C# syntax requires a data type to be specified, but no data is actually passed. For example, if no return from a method is needed, C# allows you to specify `void` as the data type instead. The declaration of `Main` within the `HelloWorld` program is an example. The use of `void` as the return type indicates that the method is not returning any data and tells the compiler not to expect a value. `void` is not a data type per se but rather an indication that there is no data being returned.

Language Contrast: C++

In both C++ and C#, `void` has two meanings: as a marker that a method does not return any data and to represent a pointer to a storage location of unknown type. In C++ programs, it is quite common to see pointer types such as `void**`. C# can also represent pointers to storage locations of unknown type using the same syntax, but this usage is comparatively rare in C# and typically encountered only when writing programs that interoperate with unmanaged code libraries.

Language Contrast: Visual Basic—Returning void Is Like Defining a Subroutine

The Visual Basic equivalent of returning a `void` in C# is to define a subroutine (`Sub`/`End Sub`) rather than a function that returns a value.

Conversions between Data Types

Given the thousands of types predefined in the various CLI implementations and the unlimited number of types that code can define, it is important that types support conversion from one type to another where it makes sense. The most common operation that results in a conversion is **casting**.

Consider the conversion between two numeric types: converting from a variable of type `long` to a variable of type `int`. A `long` type can contain values as large as 9,223,372,036,854,775,808; however, the maximum size of an `int` is 2,147,483,647. As such, that conversion could result in a loss of data—for example, if the variable of type `long` contains a value greater than the maximum size of an `int`. Any conversion that could result in a loss of magnitude or an exception because the conversion failed requires an **explicit cast**. Conversely, a conversion operation that will not lose magnitude and will not throw an exception regardless of the operand types is an **implicit conversion**.

Explicit Cast

In C#, you cast using the **cast operator**. By specifying the type you would like the variable converted to within parentheses, you acknowledge that if an explicit cast is occurring, there may be a loss of precision and data, or an exception may result. The code in Listing 2.20 converts a `long` to an `int` and explicitly tells the system to attempt the operation.

LISTING 2.20: Explicit Cast Example

```
long longNumber = 50918309109;
int intNumber = (int) longNumber;
                 Cast Operator
```

With the cast operator, the programmer essentially says to the compiler, "Trust me, I know what I am doing. I know that the value will fit into the

target type." Making such a choice will cause the compiler to allow the conversion. However, with an explicit conversion, there is still a chance that an error, in the form of an exception, might occur while executing if the data is not converted successfully. It is therefore the programmer's responsibility to ensure the data is successfully converted, or else to provide the necessary error-handling code when the conversion fails.

ADVANCED TOPIC

Checked and Unchecked Conversions

C# provides special keywords for marking a code block to indicate what should happen if the target data type is too small to contain the assigned data. By default, if the target data type cannot contain the assigned data, the data will truncate during assignment. For an example, see Listing 2.21.

Listing 2.21: Overflowing an Integer Value

```
class Program
{
  static void Main()
  {
      // int.MaxValue equals 2147483647
      int n = int.MaxValue;
      n = n + 1 ;
      System.Console.WriteLine(n);
  }
}
```

Output 2.14 shows the results.

Output 2.14

```
-2147483648
```

Listing 2.21 writes the value `-2147483648` to the console. However, placing the code within a checked block, or using the `checked` option when running the compiler, will cause the runtime to throw an exception of type `System.OverflowException`. The syntax for a checked block uses the `checked` keyword, as shown in Listing 2.22.

LISTING 2.22: A Checked Block Example

```
class Program
{
  static void Main()
  {
      checked
      {
          // int.MaxValue equals 2147483647
          int n = int.MaxValue;
          n = n + 1 ;
          System.Console.WriteLine(n);
      }
  }
}
```

Output 2.15 shows the results.

OUTPUT 2.15

```
Unhandled Exception: System.OverflowException: Arithmetic operation
resulted in an overflow at Program.Main() in ...Program.cs:line 12
```

The result is that an exception is thrown if, within the checked block, an overflow assignment occurs at runtime.

The C# compiler provides a command-line option for changing the default checked behavior from unchecked to checked. C# also supports an unchecked block that overflows the data instead of throwing an exception for assignments within the block (see Listing 2.23).

LISTING 2.23: An Unchecked Block Example

```
using System;

class Program
{
  static void Main()
  {
      unchecked
      {
          // int.MaxValue equals 2147483647
          int n = int.MaxValue;
          n = n + 1 ;
          System.Console.WriteLine(n);
      }
  }
}
```

Output 2.16 shows the results.

Output 2.16

```
-2147483648
```

Even if the checked option is on during compilation, the unchecked keyword in the preceding code will prevent the runtime from throwing an exception during execution.

Readers might wonder why, when adding 1 to `int.MaxValue` unchecked, the result yields –2147483648. The behavior is caused by wraparound semantics. The binary representation of `int.MaxValue` is 01111111111111111111111111111111, where the first digit (0) indicates a positive value. Incrementing this value yields the next value of 10000000000000000000000000000000, the smallest integer (`int.MinValue`), where the first digit (1) signifies the number is negative. Adding 1 to `int.MinValue` would result in 10000000000000000000000000000001 (–2147483647) and so on.

You cannot convert any type to any other type simply because you designate the conversion explicitly using the cast operator. The compiler will still check that the operation is valid. For example, you cannot convert a `long` to a `bool`. No such conversion is defined, and therefore, the compiler does not allow such a cast.

Language Contrast: Converting Numbers to Booleans

It may be surprising to learn that there is no valid cast from a numeric type to a Boolean type, since this is common in many other languages. The reason no such conversion exists in C# is to avoid any ambiguity, such as whether –1 corresponds to true or false. More important, as you will see in the next chapter, this constraint reduces the chance of using the assignment operator in place of the equality operator (e.g., avoiding `if(x=42){...}` when `if(x==42){...}` was intended).

Implicit Conversion

In other instances, such as when going from an `int` type to a `long` type, there is no loss of precision, and no fundamental change in the value of the type occurs. In these cases, the code needs to specify only the assignment operator; the conversion is **implicit**. In other words, the compiler is able

to determine that such a conversion will work correctly. The code in Listing 2.24 converts from an `int` to a `long` by simply using the assignment operator.

LISTING 2.24: Not Using the Cast Operator for an Implicit Conversion

```
int intNumber = 31416;
long longNumber = intNumber;
```

Even when no explicit cast operator is required (because an implicit conversion is allowed), it is still possible to include the cast operator (see Listing 2.25).

LISTING 2.25: Using the Cast Operator for an Implicit Cast

```
int intNumber = 31416;
long longNumber = (long) intNumber;
```

Type Conversion without Casting

No conversion is defined from a string to a numeric type, so methods such as `Parse()` are required. Each numeric data type includes a `Parse()` function that enables conversion from a string to the corresponding numeric type. Listing 2.26 demonstrates this call.

LISTING 2.26: Using `float.Parse()` to Convert a `string` to a Numeric Data Type

```
string text = "9.11E-31";
float kgElectronMass = float.Parse(text);
```

Another special type is available for converting one type to the next. This type is `System.Convert`, and an example of its use appears in Listing 2.27.

LISTING 2.27: Type Conversion Using `System.Convert`

```
string middleCText = "261.626";
double middleC = System.Convert.ToDouble(middleCText);
bool boolean = System.Convert.ToBoolean(middleC);
```

`System.Convert` supports only a small number of types and is not extensible. It allows conversion from any of the types `bool`, `char`, `sbyte`, `short`, `int`, `long`, `ushort`, `uint`, `ulong`, `float`, `double`, `decimal`, `DateTime`, and `string` to any other of those types.

Furthermore, all types support a `ToString()` method that can be used to provide a string representation of a type. Listing 2.28 demonstrates how to use this method. The resultant output is shown in Output 2.17.

Listing 2.28: Using `ToString()` to Convert to a `string`

```
bool boolean = true;
string text = boolean.ToString();
// Display "True"
System.Console.WriteLine(text);
```

Output 2.17

```
True
```

For the majority of types, the `ToString()` method returns the name of the data type rather than a string representation of the data. The string representation is returned only if the type has an explicit implementation of `ToString()`. One last point to make is that it is possible to code custom conversion methods, and many such methods are available for classes in the runtime.

Begin 2.0

ADVANCED TOPIC

TryParse()

Starting with C# 2.0 (.NET 2.0), all the numeric primitive types include a static `TryParse()` method. This method is similar to the `Parse()` method, except that instead of throwing an exception if the conversion fails, the `TryParse()` method returns `false`, as demonstrated in Listing 2.29.

Listing 2.29: Using `TryParse()` in Place of an Invalid Cast Exception

```
double number;
string input;

System.Console.Write("Enter a number: ");
input = System.Console.ReadLine();
if (double.TryParse(input, out number))
{
    // Converted correctly, now use number
    // ...
}
```

```
else
{
    System.Console.WriteLine(
        "The text entered was not a valid number.");
}
```

Output 2.18 shows the results of Listing 2.29.

OUTPUT 2.18

```
Enter a number: forty-two
The text entered was not a valid number.
```

The resultant value that the code parses from the input `string` is returned via an `out` parameter—in this case, `number`.

It is worth pointing out that, starting with C# 7.0, it is no longer necessary to declare a variable before using it as an `out` argument. Using this feature, the declaration for number is shown in Listing 2.30.

Begin 7.0

LISTING 2.30: Using `TryParse()` with Inline `out` Declaration in C# 7.0

```
// double number;
string input;

System.Console.Write("Enter a number: ");
input = System.Console.ReadLine();
if (double.TryParse(input, out double number))
{
    System.Console.WriteLine(
        $"input was parsed successfully to {number}."); }
else
{
    // Note: number scope is here too (although not assigned)
    System.Console.WriteLine(
        "The text entered was not a valid number.");
}
```

Notice that the data type of `number` is specified following the `out` modifier and before the variable that it declares. The result is that the `number` variable is available from both the true and false consequence of the `if` statement but not outside the `if` statement.

End 7.0

The key difference between `Parse()` and `TryParse()` is that `TryParse()` won't throw an exception if it fails. Frequently, the conversion from a `string` to a numeric type depends on a user entering the text. It is

2.0

expected, in such scenarios, that the user will enter invalid data that will not parse successfully. By using `TryParse()` rather than `Parse()`, you can avoid throwing exceptions in expected situations. (The expected situation in this case is that the user will enter invalid data, and we try to avoid throwing exceptions for expected scenarios.)

End 2.0

SUMMARY

Even for experienced programmers, C# introduces several new programming constructs. For example, as part of the section on data types, this chapter covered the type `decimal`, which can be used to perform financial calculations without floating-point anomalies. In addition, the chapter introduced the fact that the Boolean type, `bool`, does not convert implicitly to or from the integer type, thereby preventing the mistaken use of the assignment operator in a conditional expression. Other characteristics of C# that distinguish it from many of its predecessors are the `@` verbatim string qualifier, which forces a string to ignore the escape character, string interpolation that makes code easier to read by embedding it into the string, and the immutable nature of the `string` data type.

In Chapter 3, we continue the topic of data types by elaborating more on the two types of data types: value types and reference types. In addition, we look at combining data elements together into tuples and arrays.